B-Trees

Definition

An order m B-tree is a search tree such that each interior node has up to m children. The *values* of the data are stored in the leaves while the interior nodes only contain *keys*.

Wikipedia quotes Knuth as defining a B-Tree as an order m tree which satisfies the following properties that I will paraphrase as:

1. Every node has at most m children (where m is the *order* of the tree.)
2. Every non-leaf node (except root) has at least $[m/2]$ child nodes.
3. The root has at least two children if it is not a leaf node.
4. A non-leaf node with k children contains $k-1$ keys.
5. All leaves appear in the same level.[1]

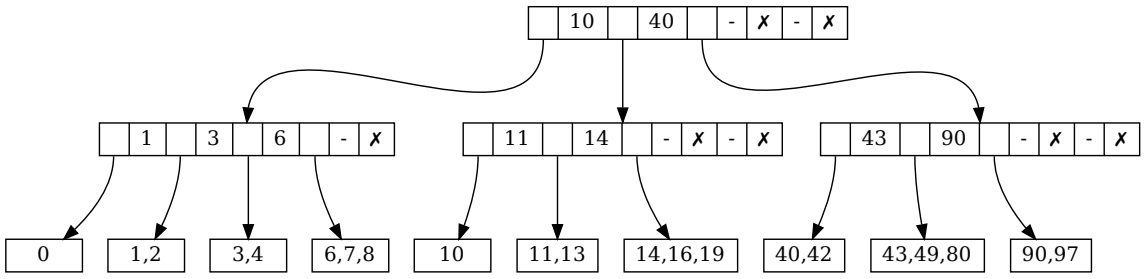

Figure 1: B-Tree, Order=5

This structure will achieve O(log n) performance.

Why Do We Care About This?

B-trees can be useful for storing data on disk where the time to access the data can be several orders of magnitude slower than accessing a machine's main memory.

The idea is that building a tree such that a search requires the need to visit the least number of nodes possible. By increasing the number of children for a given node the height of the tree can be reduced (while the width is increased.) *Recall that the number of nodes visited during a search is O(tree-height)!*

Optimally, the choice of the order of B-Tree is typically defined to minimize the number of disk accesses by preventing the nodes in the tree from spanning more than one block of disk!

[1] Wikipedia literally says: *All leaves appear in the same level and carry no information.* But that makes no sense... at least, to me.

~/NIU/courses/340/2021-sp/slides/b-tree/btree.tex
jwinans@niu.edu 2021-04-14 20:00:16 -0500 v2.0-1074-gd430f86

Searching

Searching can be done by using a linear (or binary) search of the keys in a given node to locate the key or determine which child node to traverse to.

Insertion

Let us assume that our order-5 tree can contain up to three elements in each of its leaf nodes. Optimally, this number would be determined by how many elements are expected to fit into a single disk block.

For insertion, search for the leaf node as above to locate the leaf node that should contain the new element.

If there is room in the leaf for the the new element then add it to the leaf.

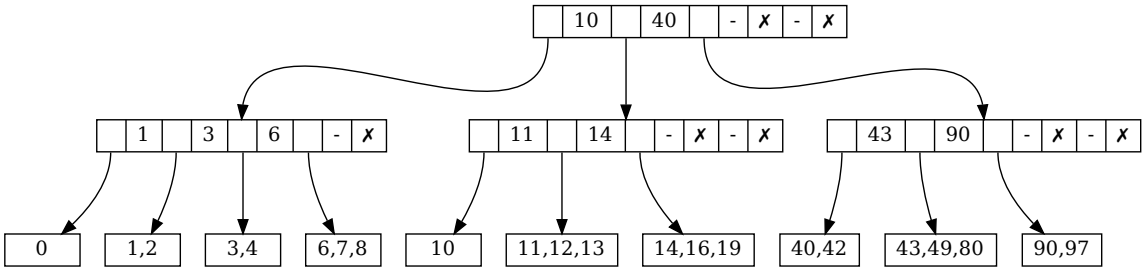

Figure 2: Inserting an element with key=12

Else split the leaf into two where one will contain the new element and update the parent node by adding the new child pointer and key.

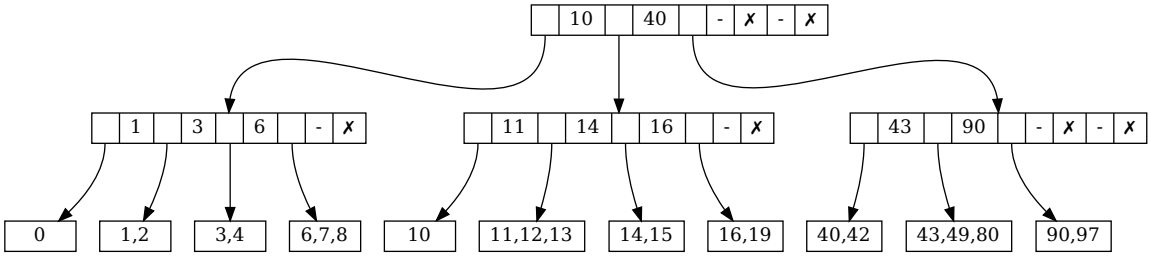

Figure 3: Inserting an element with key=15

If the is no room in the parent node then the parent will be split in two and recursively continue upwards as far as necessary. If the root is full and needs splitting in two as well then a new root node will be created that points to the two new children (from the old root-split.) Observing that the height would have been increased by one.

Deletion

To delete a node, the opposite operation would be performed... combining neighboring nodes should any one ever become too empty.

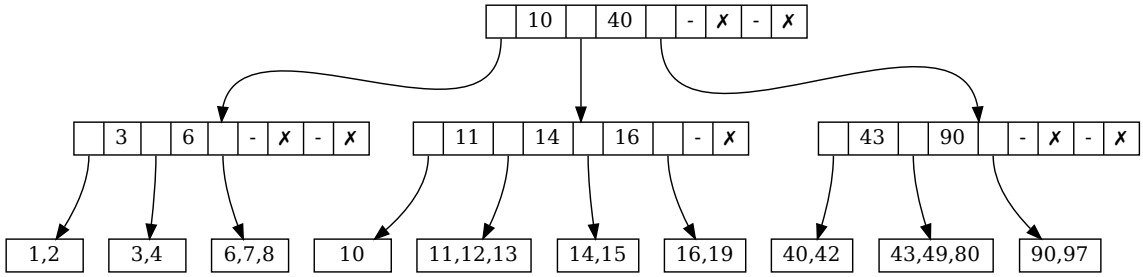

Figure 4: Removing the element with key=0

One way to handle the removal of keys 3 and 4 while optimizing for minimum space would require the restructuring of the leaves and interior nodes as shown in Figure 5.

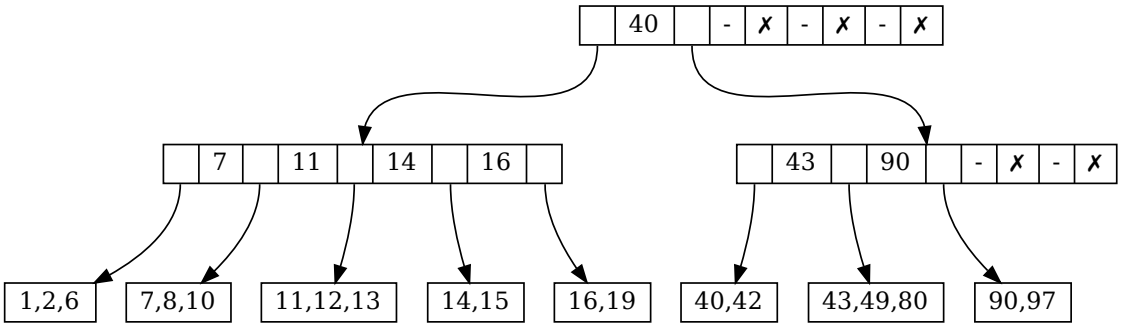

Figure 5: Removing the elements with keys 3 and 4 while optimizing for space

A different way to handle the removal of keys 3 and 4 while optimizing for the ease of future insertions (by keeping some empty space in the leaves) is shown in Figure 6.

Observe that if the root node is reduced to having only one child, it will be removed and its child will become the new root leaving the height of the tree one less than before.

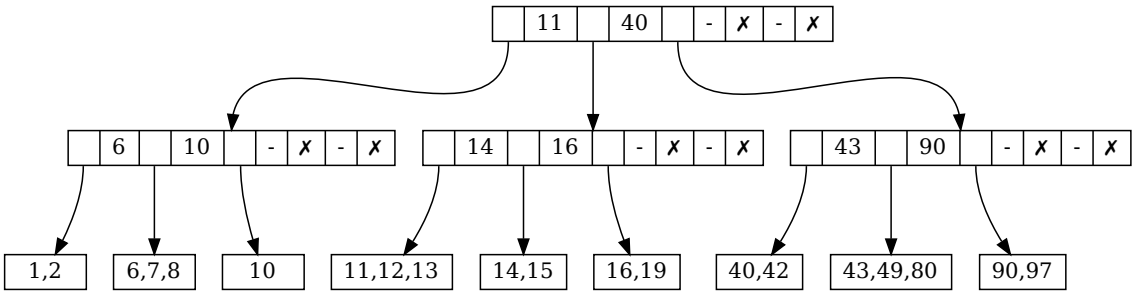

Figure 6: Removing the elements with keys 3 and 4 while optimizing for future growth